



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2025/26)

Lic. em Ciências da Computação

Lic. em Engenharia Informática

Grupo G05

a106936 Duarte Escairo

a106932 Luís Soares

a106856 Tiago Figueiredo

Preâmbulo

Em [Cálculo de Programas](#) pretende-se ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em [Haskell](#) (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em [Haskell](#). Há ainda um outro objectivo: o de ensinar a documentar programas, a validá-los e a produzir textos técnico-científicos de qualidade.

Antes de abordarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao *software* a instalar, etc.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Avaliação. Faz parte da avaliação do trabalho a sua defesa por parte dos elementos de cada grupo. Estes devem estar preparados para responder a perguntas sobre *qualquer* dos problemas deste enunciado. A prestação *individual* de cada aluno nessa defesa oral será uma componente importante e diferenciadora da avaliação.

Problema 1

Uma serialização (ou travessia) de uma árvore é uma sua representação sob a forma de uma lista. Na biblioteca *BTree* encontram-se as funções de serialização *inordt*, *preordt* e *postordt*, que fazem as travessias *in-order*, *pre-order* e *post-order*, respectivamente. Todas essas travessias são catamorfismos que percorrem a árvore argumento em regime *depth-first*.

Pretende-se agora uma função *bforder* que faça a travessia em regime *breadth-first*, isto é, por níveis. Por exemplo, para a árvore t_1 dada em anexo e mostrada na figura a seguir,



a função deverá dar a lista

[5, 3, 7, 1, 4, 6, 8]

em que se vê como os níveis 5, depois 3, 7 e finalmente 1, 4, 6, 8 foram percorridos.

Pretendemos propor duas versões dessa função:

1. Uma delas envolve um catamorfismo de *BTrees*:

```
bfsLevels :: BTTree a → [a]  
bfsLevels = concat · levels
```

Complete a definição desse catamorfismo:

```
levels :: BTTree a → [[a]]  
levels = (λ glevels)
```

2. A segunda proposta,

```
bft :: BTTree a → [a]
```

deverá basear-se num anamorfismo de listas.

Sugestão: estudar o artigo [2] cujo PDF está incluído no material deste trabalho. Quando fizer testes ao seu código pode, se desejar, usar funções disponíveis na biblioteca *Exp* para visualizar as árvores em GraphViz (formato .dot).

Justifique devidamente a sua resolução, que deverá vir acompanhada de diagramas explicativos. Como já se disse, valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Problema 2

Considere a seguinte função em Haskell:

```
f x = wrapper · worker where  
    wrapper = head  
    worker 0 = start x  
    worker (n + 1) = loop x (worker n)  
    loop x [s, h, k, j, m] =  
        [h / k + s, x ↑ 2 * h, k * j, j + m, m + 8]  
    start x = [x, x ↑ 3, 6, 20, 22]
```

Pode-se provar pela lei de recursividade mútua que $f x n$ calcula o seno hiperbólico de x , $\sinh x$, para n aproximações da sua série de Taylor. Faça a derivação da função dada a partir da referida série de Taylor, apresentando todos os cálculos justificativos, tal como se faz para outras funções no capítulo respectivo do texto base desta UC [3].

Problema 3

Quem em Braga observar, ao fim da tarde, o tráfego onde a Avenida Clairmont Fernand se junta à N101, aproximadamente na coordenada [41°33'46.8"N 8°24'32.4"W](#) – ver as setas da figura que se segue – reparará nas sequências imparáveis (infinitas!) de veículos provenientes dessas vias de circulação.

Mas também irá observar um comportamento interessante por parte dos condutores desses veículos: por regra, *cada carro numa via deixa passar, à sua frente, exactamente outro carro da outra via*.



Este comportamento *civilizado* chama-se *fair-merge* (ou *fair-interleaving*) de duas sequências infinitas, também designadas *streams* em ciência da computação. Seja dado o tipo dessas sequências em Haskell,

```
data Stream a = Cons (a, Stream a) deriving Show
```

para o qual se define também:

```
out (Cons (x, xs)) = (x, xs)
```

O referido comportamento civilizado pode definir-se, em Haskell, da forma seguinte:¹

```
fair_merge :: (Stream a, Stream a) + (Stream a, Stream a) → Stream a
fair_merge = [h, k] where
  h (Cons (x, xs), y) = Cons (x, k (xs, y))
  k (x, Cons (y, ys)) = Cons (y, h (x, ys))
```

Defina *fair_merge* como um **anamorfismo** de *Streams*, usando o combinador

```
[(g)] = Cons · (id × [(g)]) · g
```

e a seguinte estratégia:

- Derivar a lei **dual** da recursividade mútua,

$$[f, g] = [(h, k)] \equiv \begin{cases} out \cdot f = F[f, g] \cdot h \\ out \cdot g = F[f, g] \cdot k \end{cases} \quad (1)$$

tal como se fez, nas aulas, para a que está no formulário.

- Usar (1) na resolução do problema proposto.

Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Problema 4

Como se sabe, é possível pensarmos em catamorfismos, anamorfismos etc *probabilísticos*, quer dizer, programas recursivos que dão distribuições como resultados. Por exemplo, podemos pensar num combinador

```
pcataList :: ((()) + (a, b) → Dist b) → [a] → Dist b
```

¹ O facto das sequências serem infinitas não nos deve preocupar, pois em Haskell isso é lidado de forma transparente por [lazy evaluation](#).

que é muito parecido com

$$(\cdot) :: ((a, b) \rightarrow b) \rightarrow [a] \rightarrow b$$

da biblioteca [List](#). A principal diferença é que o gene de *pcataList* é uma função probabilística.

Como exemplo de utilização, recorde-se que $([zero, add])$ soma todos os elementos da lista argumento, por exemplo:

$$([zero, add]) [20, 10, 5] = 35.$$

Considere-se agora a função *padd* (adição probabilística) que, com probabilidade 90% soma dois números e com probabilidade 10% os subtrai:

$$padd(a, b) = D[(a + b, 0.9), (a - b, 0.1)]$$

Se se correr

$$d4 = pcataList [pzero, padd] [20, 10, 5] \text{ where } pzero = return \cdot zero$$

obter-se-á:

35	81.0%
25	9.0%
5	9.0%
15	1.0%

Com base neste exemplo, resolva o seguinte

Problema: Uma unidade militar pretende enviar uma mensagem urgente a outra, mas tem o aparelho de telegrafia meio avariado. Por experiência, o telegrafista sabe que a probabilidade de uma palavra se perder (não ser transmitida) é 5%; e que, no final de cada mensagem, o aparelho envia o código "stop", mas (por estar meio avariado), falha 10% das vezes.

Qual a probabilidade de a palavra "atacar" da mensagem

words "Vamos atacar hoje"

se perder, isto é, o resultado da transmissão ser ["Vamos", "hoje", "stop"]? E a de seguirem todas as palavras, mas faltar o "stop" no fim? E a da transmissão ser perfeita?

Responda a estas perguntas encontrando gene tal que

transmitir = *pcataList* *gene*

descreve o comportamento do aparelho. Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na [página da disciplina](#) na internet.

Recomenda-se uma abordagem participativa dos membros do grupo em **todos** os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [1], cujo princípio base é o seguinte:

Um programa e a sua documentação devem coincidir.

Por outras palavras, o **código fonte** e a **documentação** de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp2526t.pdf` que está a ler é já um exemplo de [programação literária](#): foi gerado a partir do texto fonte `cp2526t.lhs`¹ que encontrará no [material pedagógico](#) desta disciplina descompactando o ficheiro `cp2526t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código [Haskell](#) que ele inclui:



Vê-se assim que, para além do [GHCi](#), serão necessários os executáveis [pdflatex](#) e [lhs2TeX](#). Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do [Docker](#) tal como a seguir se descreve.

B Docker

Recomenda-se o uso do [container](#) cuja imagem é gerada pelo [Docker](#) a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2526t.zip`. Este [container](#) deverá ser usado na execução do [GHCi](#) e dos comandos relativos ao [LATEX](#). (Ver também a `Makefile` que é disponibilizada.)

Após [instalar o Docker](#) e descargar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2526t .
$ docker run -v ${PWD}:/cp2526t -it cp2526t
```

NB: O objetivo é que o container seja usado *apenas* para executar o [GHCi](#) e os comandos relativos ao [LATEX](#). Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2526t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2526t` no [container](#) sejam partilhadas.

Pretende-se então que visualize/editie os ficheiros na sua máquina local e que os compile no [container](#), executando:

¹ O sufixo ‘lhs’ quer dizer *literate Haskell*.

```
$ lhs2TeX cp2526t.lhs > cp2526t.tex  
$ pdflatex cp2526t
```

[lhs2TeX](#) é o pre-processador que faz “pretty printing” de código Haskell em [LaTeX](#) e que faz parte já do [container](#). Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro cp2526t.lhs é executável e contém o “kit” básico, escrito em [Haskell](#), para realizar o trabalho. Basta executar

```
$ ghci cp2526t.lhs
```

Abra o ficheiro cp2526t.lhs no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}  
...  
\end{code}
```

é seleccionado pelo [GHCi](#) para ser executado.

C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo G com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com [BibTeX](#)) e o índice remissivo (com [makeindex](#)),

```
$ bibtex cp2526t.aux  
$ makeindex cp2526t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente [make](#) no [container](#).)

No anexo F disponibiliza-se algum código [Haskell](#) relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da [programação literária](#) para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo D que se segue.

D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte ([lhs](#)) do que está a ler¹ onde se obtém o efeito seguinte:²

$$\begin{aligned} id &= \langle f, g \rangle \\ &\equiv \{ \text{ universal property } \} \end{aligned}$$

¹ Procure e.g. por "sec:diagramas".

² Exemplos tirados de [3].

$$\begin{aligned}
 & \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
 \equiv & \quad \{ \text{identity} \} \\
 & \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
 \square
 \end{aligned}$$

Os diagramas podem ser produzidos recorrendo à package [xymatrix](#), por exemplo:

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \downarrow (g) & & \downarrow id + (g) \\
 B & \xleftarrow[g]{\quad} & 1 + B
 \end{array}$$

E O mónade das distribuições probabilísticas

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca [Probability](#) oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

newtype Dist $a = D \{ unD :: [(a, ProbRep)] \}$ (2)

em que $ProbRep$ é um real de 0 a 1, equivalente a uma escala de 0 a 100%.

Cada par (a, p) numa distribuição $d :: \text{Dist } a$ indica que a probabilidade de a é p , devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E ,



será representada pela distribuição

```
d1 :: Dist Char
d1 = D [('A', 0.02), ('B', 0.12), ('C', 0.29), ('D', 0.35), ('E', 0.22)]
```

que o [GHCi](#) mostrará assim:

```
'D' 35.0%
'C' 29.0%
'E' 22.0%
'B' 12.0%
'A' 2.0%
```

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

```
d2 = uniform (words "Uma frase de cinco palavras")
```

isto é

```

"Uma" 20.0%
"cinco" 20.0%
"de" 20.0%
"frase" 20.0%
"palavras" 20.0%

```

distribuição *normais*, eg.

$$d3 = \text{normal}[10..20]$$

etc.¹ Dist forma um **mónade** cuja unidade é $\text{return } a = D[(a, 1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g) a = [(y, q * p) \mid (x, p) \leftarrow g a, (y, q) \leftarrow f x]$$

em que $g : A \rightarrow \text{Dist } B$ e $f : B \rightarrow \text{Dist } C$ são funções **monádicas** que representam *computações probabilísticas*.

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular da programação monádica.

F Código fornecido

Problema 1

Árvores exemplo:

```

t1 :: BTTree Int
t1 = Node(5, (Node(3, (Node(1, (Empty, Empty)), Node(4, (Empty, Empty)))), Node(7, (Node(6, (Empty, Empty)), Node(8, (Empty, Empty))))))

t2 :: BTTree Int
t2 =
  node 1
    (node 2 (node 4 Empty Empty) (node 5 Empty Empty))
    (node 3 (node 6 Empty Empty) (node 7 Empty Empty))

t3 :: BTTree Char
t3 =
  node 'A'
    (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
    (node 'E' Empty Empty)

t4 :: BTTree Char
t4 =
  node 'A'
    (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
    Empty

t5 :: BTTree Int
t5 =
  node 1

```

¹ Para mais detalhes ver o código fonte de [Probability](#), que é uma adaptação da biblioteca [PFP](#) (“Probabilistic Functional Programming”). Para quem quiser saber mais recomenda-se a leitura do artigo [?].

$$\begin{aligned}
 & (\text{node } 2 (\text{node } 4 \text{ Empty Empty}) \text{ Empty}) \\
 & (\text{node } 3 \text{ Empty } (\text{node } 5 (\text{node } 6 \text{ Empty Empty}) \text{ Empty})) \\
 & \text{node } a b c = \text{Node } (a, (b, c))
 \end{aligned}$$

G Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o “layout” que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

Importante: Não pode ser alterado o texto deste ficheiro fora deste anexo.

Problema 1

Catamorfismo

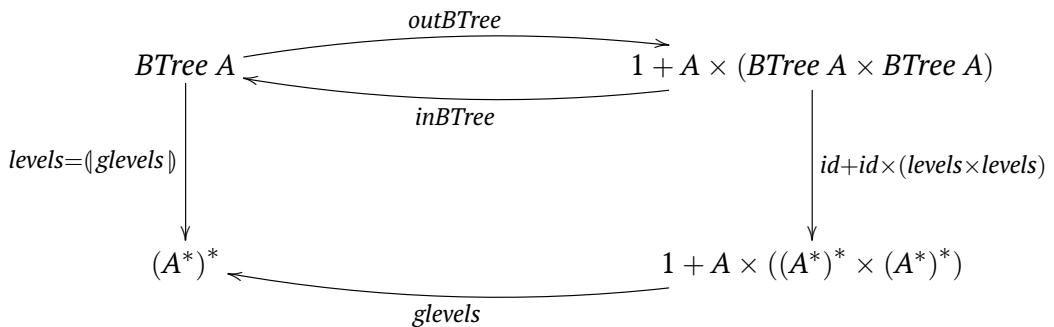
Na primeira versão proposta para a resolução do **Problema 1**, pretende-se usar um catamorfismo de *BTrees* para se fazer a travessia *in-order* em regime *breadth-first*.

Se repararmos, o resultado de aplicarmos a função *levels* a uma *BTree* é uma lista de listas, onde cada uma dessas listas internas corresponde aos valores dos nós de um nível da árvore. Ou seja, a aplicação de *levels* à árvore t_1 , por exemplo, resulta na lista de listas:

$$[[5], [3, 7], [1, 4, 6, 8]]$$

Para depois obter a travessia *bftorder*, basta concatenar todas as listas internas, o que é feito na função *bfsLevels* com recurso à função *concat*.

O desafio aqui está em encontrar o gene (*glevels*) do catamorfismo *levels*. Para começar, podemos representar esse catamorfismo através do seguinte diagrama:



A partir deste diagrama, percebemos que o gene do catamorfismo deverá, para obter a tal lista de listas, colocar o valor da raiz no início da lista final, seguido das listas dos níveis das subárvores esquerda e direita. Contudo essas listas dos níveis das subárvores estão também organizadas por níveis, ou seja, a primeira lista corresponde ao nível 1, a segunda ao nível 2, etc. No caso da árvore t_1 , por exemplo, a aplicação de *levels* às subárvores esquerda e direita de t_1 resulta, respectivamente, nas listas de listas:

$$[[3], [1, 4]] \text{ e } [[7], [6, 8]]$$

Neste caso o passo final seria juntar as listas dos níveis das subárvores esquerda e direita, e colocar o valor da raiz 5 ([5]) no início da lista final. Para juntar as listas dos níveis das subárvores esquerda e direita, podemos definir a seguinte função auxiliar:

$$\begin{aligned} juntaListas :: ([[a]], [[a]]) &\rightarrow [[a]] \\ juntaListas(l, []) &= l \\ juntaListas([], r) &= r \\ juntaListas((a : as), (b : bs)) &= (a ++ b) : juntaListas(as, bs) \end{aligned}$$

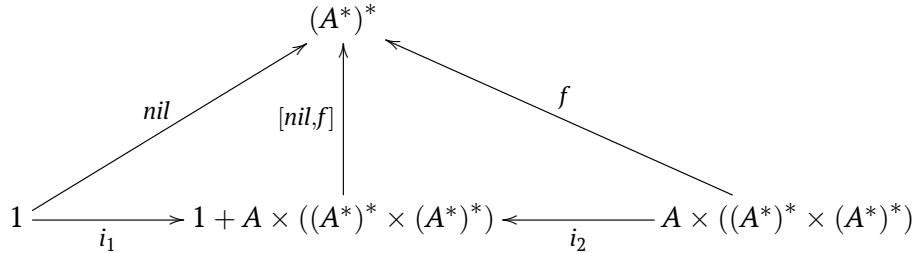
Para o exemplo da árvore t_1 , a aplicação de $juntaListas$ às listas de listas

$[[3], [1, 4]]$ e $[[7], [6, 8]]$

resulta na lista de listas:

$[[3, 7], [1, 4, 6, 8]]$

Assim sendo, o comportamento do gene $glevels$ pode ser representado por este diagrama:

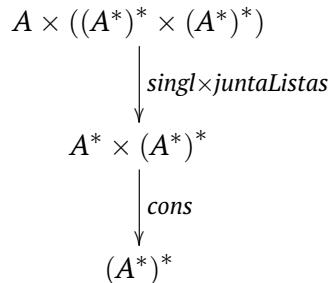


E o código que define o gene $glevels$ é o seguinte:

$$\begin{aligned} glevels :: () + (a, ([[a]], [[a]])) &\rightarrow [[a]] \\ glevels = [nil, f] \\ \text{where } f(a, (l, r)) &= [a] : juntaListas(l, r) \end{aligned}$$

Tal como descrito anteriormente, f coloca o valor da raiz à cabeça da lista, seguido da concatenação das listas dos níveis das subárvores esquerda e direita, com a função $juntaListas$.

Se quisermos definir $glevels$ numa versão completamente *pointfree*, podemos esquematizar o lado direito da solução desta forma:



Logo a definição *pointfree* de $glevels$ é a seguinte:

$glevels' = [nil, cons \cdot (singl \times juntaListas)]$

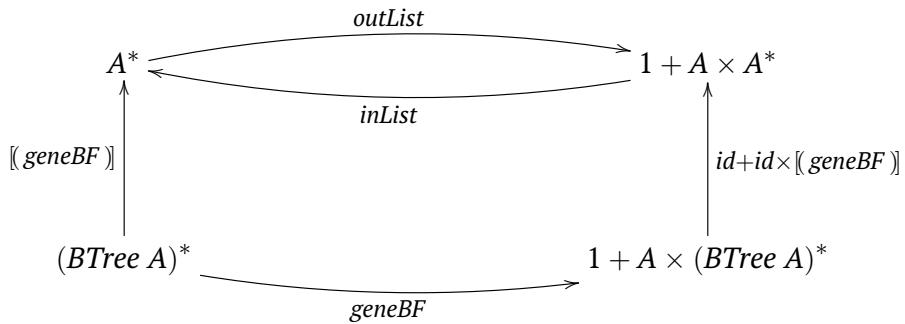
Anamorfismo

Na segunda versão proposta para a resolução do **Problema 1**, pretende-se usar um anamorfismo de listas para se fazer a travessia *in-order* em regime *breadth-first*.

Com o estudo do artigo [2], percebemos que é possível fazer a travessia *breadth-first* em *BTrees* usando uma floresta para representar os nós a visitar (neste caso uma floresta é uma lista de *BTrees*).

A ideia do algoritmo é visitar os nós da floresta, retirando o valor do nó da frente da lista, adicionando o seu valor à lista resultado (caso não seja *Empty*) e adicionando à floresta os filhos esquerdo e direito do nó visitado (caso não seja *Empty*), mas colocando-os no final da lista. Este processo repete-se até que a floresta esteja vazia.

Para representar este algoritmo como um anamorfismo, começamos por desenhar o seguinte diagrama:



Contudo, como referido anteriormente, o algoritmo usa uma floresta (lista de *BTrees*), e não um *BTree*, que é o input da função *bft*. Desta forma é necessário garantir que a *BTree* argumento seja convertida numa floresta, para poder ser usada no anamorfismo. Para isso, usamos a função *singl*, que transforma uma *BTree* numa floresta com apenas essa *BTree* (lista com apenas um elemento).

$$BTree\ A \xrightarrow{\text{singl}} (BTree\ A)^*$$

Assim sendo, a função *bft* é definida como a composição do anamorfismo $[(geneBF)]$ com a função *singl*, como se mostra a seguir:

$$bft = [(geneBF)] \cdot singl$$

Para o gene do anamorfismo, o comportamento é o já anteriormente descrito, ou seja, visitar os nós da floresta, retirando o valor do nó da frente da lista, e devolvendo um par com o valor do nó e a floresta a visitar. Na floresta a visitar, já estarão adicionados os filhos esquerdo e direito do nó visitado (caso não seja *Empty*), colocados no final da lista. O anamorfismo repete este processo até que a floresta esteja vazia.

$$\begin{aligned} geneBF :: [BTree\ a] &\rightarrow () + (a, [BTree\ a]) \\ geneBF [] &= i_1 () \\ geneBF (Empty : t) &= geneBF\ t \\ geneBF (Node\ (a, (l, r)) : t) &= i_2\ (a, t ++ [l, r]) \end{aligned}$$

Problema 2

No **Problema 2**, o objetivo é derivarmos a função f , que calcula o seno hiperbólico de x , $\sinh x$, para n aproximações da sua série de Taylor, a partir da série de Taylor de $\sinh x$ que é dada por:

$$\sinh x = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots$$

No nosso caso, iremos utilizar uma aproximação com n termos da série de Taylor, ou seja:

$$\sinh x n = \sum_{i=0}^{n-1} \frac{x^{2i+1}}{(2i+1)!}$$

A derivação para a função f é necessária pois no caso de definirmos $\sinh x n$ diretamente de forma recursiva

$$\begin{aligned}\sinh x 0 &= x \\ \sinh x (n+1) &= (x \uparrow (2 * (n+1) + 1)) / (2 * (n+1) + 1)! + \sinh x n\end{aligned}$$

e fazendo os cálculos temos

$$\begin{aligned}\sinh x 0 &= x \\ \sinh x (n+1) &= (x \uparrow (2 * n + 3)) / (2 * n + 3)! + \sinh x n\end{aligned}$$

verificamos que esta definição não é eficiente, já que em cada passo recursivo é necessário calcular $x \uparrow (2 * n + 3)$ e $(2 * n + 3)!$, o que não se verifica em f , pois com o *loop*, os valores do vetor inicial $start x = [x, x \uparrow 3, 6, 20, 22]$ são atualizados em cada iteração de forma linear, evitando cálculos desnecessários.

Com o estudo da secção da Lei da Recursividade Mútua no livro do professor J.N.Oliveira [3], entendemos que devíamos partir o problema em duas funções recursivas, uma referente ao numerador da fração de $\sinh x n$ e outra referente ao denominador, de forma a podermos aplicar a lei da recursividade mútua. Assim sendo, poderíamos definir $\sinh x n$ como uma soma da seguinte forma:

$$\begin{aligned}soma x 0 &= x \\ soma x (n+1) &= num x (n+1) / den x (n+1) + soma x n\end{aligned}$$

onde $num x n$ calcula o numerador $x \uparrow (2 * n + 1)$ e $den x n$ calcula o denominador $(2 * n + 1)!$.

Para o numerador, podemos descobrir qual é a diferença entre dois termos consecutivos fazendo a seguinte análise:

$$\frac{num_{n+1}}{num_n} = \frac{x^{2(n+1)+3}}{x^{2n+3}} = \frac{x^{2n+5}}{x^{2n+3}} = x^2$$

ou seja, o próximo termo do numerador é obtido multiplicando o termo atual por x^2 . Podemos concluir que o numerador pode ser definido recursivamente da seguinte forma:

$$\begin{aligned}num x 1 &= x \uparrow 3 \\ num x (n+1) &= x \uparrow 2 * num x n\end{aligned}$$

Para o denominador, podemos descobrir qual é a diferença entre dois termos consecutivos fazendo a seguinte análise:

$$\frac{den_{n+1}}{den_n} = \frac{(2(n+1)+3)!}{(2n+3)!} = \frac{(2n+5)!}{(2n+3)!} = (2n+5)(2n+4)$$

ou seja, o próximo termo do denominador é obtido multiplicando o termo atual por $(2n+5)(2n+4)$. Podemos concluir que o denominador pode ser definido recursivamente da seguinte forma:

$$\begin{aligned} \text{den } x \ 1 &= 3! = 6 \\ \text{den } x \ (n+1) &= (2n+5)(2n+4) * \text{den } x \ n \end{aligned}$$

Contudo, esta definição recursiva do denominador ainda depende de n , o que não é desejável, por isso, é necessário a simplificar, até não depender de n . Para isso, faça-se

$$j \ x \ n = (2n+5) * (2n+4)$$

tal como é feito no livro do professor.

Para descobrirmos a diferença entre dois termos consecutivos, fazemos a seguinte análise:

$$j_{n+1} - j_n = (2(n+1)+5)(2(n+1)+4) - (2n+5)(2n+4) = 8n + 22$$

ou seja, o próximo termo de j é obtido somando ao termo atual $8n + 22$. Podemos concluir que j pode ser definida recursivamente da seguinte forma:

$$\begin{aligned} j \ x \ 0 &= 20 \\ j \ x \ (n+1) &= (8 * n + 22) + j \ x \ n \end{aligned}$$

Mais uma vez, esta definição recursiva de j ainda depende de n , o que não é desejável, por isso, voltamos a repetir o processo até não depender de n . Para isso, faça-se

$$m \ x \ n = 8 * n + 22$$

Para descobrirmos a diferença entre dois termos consecutivos, fazemos a seguinte análise:

$$m_{n+1} - m_n = 8(n+1) + 22 - (8n + 22) = 8$$

ou seja, o próximo termo de m é obtido somando ao termo atual 8. Podemos concluir que m pode ser definida recursivamente da seguinte forma:

$$\begin{aligned} m \ x \ 0 &= 22 \\ m \ x \ (n+1) &= 8 + m \ x \ n \end{aligned}$$

Agora que m tem uma definição recursiva que não depende de n , podemos aplicar a lei da recursividade mútua, de forma a obter a definição de f .

Recapitulando todas as funções que foram definidas nesta derivação temos:

$$\begin{aligned} \text{soma } x \ 0 &= x \\ \text{soma } x \ (n+1) &= \text{num } x \ (n+1) / \text{den } x \ (n+1) + \text{soma } x \ n \end{aligned}$$

$$\begin{aligned} \text{num } x \ 1 &= x \uparrow 3 \\ \text{num } x \ (n+1) &= x \uparrow 2 * \text{num } x \ n \end{aligned}$$

$$\begin{aligned} \text{den } x \ 1 &= 3! = 6 \\ \text{den } x \ (n+1) &= (2n+5)(2n+4) * \text{den } x \ n \end{aligned}$$

$$\begin{aligned} j \ x \ 0 &= 20 \\ j \ x \ (n+1) &= (8 * n + 22) + j \ x \ n \end{aligned}$$

$$\begin{aligned} m \times 0 &= 22 \\ m \times (n+1) &= 8 + m \times n \end{aligned}$$

E com estas funções, podemos reconstruir *soma* da seguinte forma:

$$\begin{aligned} m \times 0 &= 22 \\ m \times (n+1) &= 8 + m \times n \end{aligned}$$

$$\begin{aligned} j' \times 0 &= 20 \\ j' \times (n+1) &= m \times n + j' \times n \end{aligned}$$

$$\begin{aligned} den' \times 1 &= 3! = 6 \\ den' \times (n+1) &= j' \times n * den' \times n \end{aligned}$$

$$\begin{aligned} soma' \times 0 &= x \\ soma' \times (n+1) &= num \times (n+1) / den' \times (n+1) + soma' \times n \end{aligned}$$

E pela lei da recursividade mútua, podemos escrever *soma* como:

$$\begin{aligned} soma'' \times n &= s \text{ where} \\ (s, -, -, -, -) &= aux \times 0 \\ aux \times 0 &= (x, x \uparrow 3, 6, 20, 22) \\ aux \times (n+1) &= \text{let } (s, num, den, j', m) = aux \times n \text{ in } (s + num / den, x \uparrow 2 * num, den * j', j' + m, m + 8) \end{aligned}$$

Tal como é dito no livro, *aux* é um *loop*, onde *aux* 0 corresponde ao valor inicial e *aux* ($n+1$) corresponde à atualização dos valores em cada iteração do *loop*.

Desta forma, adaptando a notação usada no livro do professor, conseguimos definir *start x* e *loop x* como:

$$start x = [x, x \uparrow 3, 6, 20, 22]$$

$$loop x [s, num, den, j', m] = [s + num / den, x \uparrow 2 * num, den * j', j' + m, m + 8]$$

Trocando o nome de algumas funções, temos o *loop* como:

$$loop x [s, h, k, j', m] = [s + h / k, x \uparrow 2 * h, k * j, j + m, m + 8]$$

Na variável *s* está o valor acumulado da soma, ou seja, o cálculo de *sinh x n*, pelo que basta fazer *head* do resultado do *loop* após *n* iterações, para obter o valor final de *sinh x n*.

Fica assim derivada a função *f* como pedido no enunciado.

Problema 3

No **Problema 3**, pretende-se definir a função *fair_merge* como um anamorfismo de *Streams*. Esta estratégia de *merging* permite que os elementos das duas *Streams* sejam intercalados de forma justa, ou seja, cada elemento de uma *Stream* é seguido por um elemento da outra *Stream*.

Tal como pedido no enunciado, vamos começar por derivar a lei dual da recursividade mútua.

$$\begin{aligned}
& [f, g] = [[h, k]] \\
\equiv & \quad \{ \text{Universal-ana} \} \\
& \text{out} \cdot [f, g] = F[f, g] \cdot [h, k] \\
\equiv & \quad \{ \text{Fusão- + (2x)} \} \\
& [\text{out} \cdot f, \text{out} \cdot g] = [F[f, g] \cdot h, F[f, g] \cdot k] \\
\equiv & \quad \{ \text{Eq- + } \} \\
& \left\{ \begin{array}{l} \text{out} \cdot f = F[f, g] \cdot h \\ \text{out} \cdot g = F[f, g] \cdot k \end{array} \right. \\
\square
\end{aligned}$$

Fica então demonstrada a lei dual da recursividade mútua.

A partir da definição do tipo *Stream* e da função *out*, é possível deduzir que o functor *F* associado a *Stream* é o seguinte,

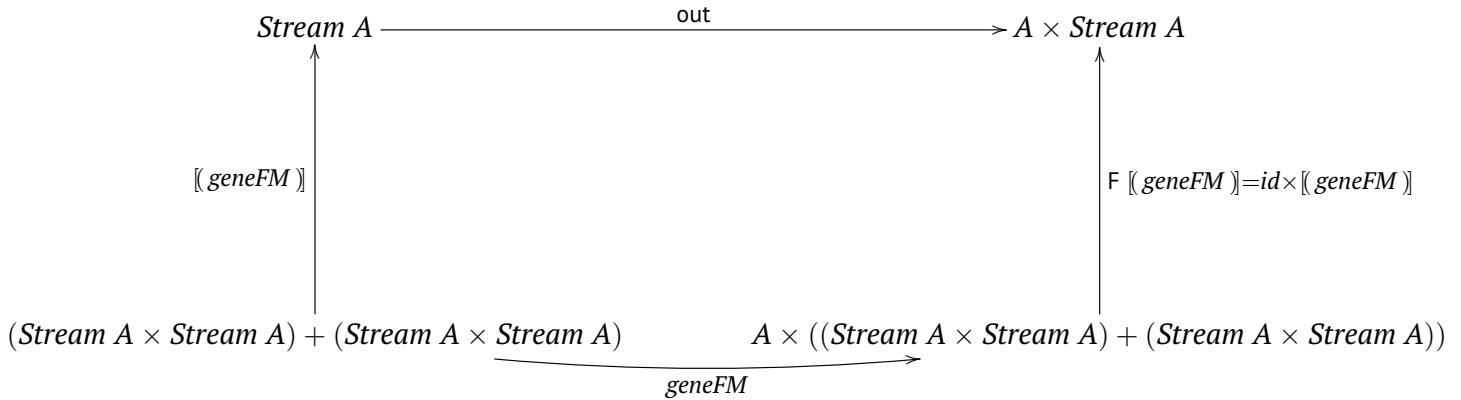
$$\begin{aligned}
F X &= A \times X \\
F g &= id \times g
\end{aligned}$$

já que uma *Stream A* é decomposta num par com o primeiro elemento de um tipo qualquer *A* e o resto da *Stream A*.

Tendo isto em conta, e sabendo que a função *fair_merge'* que queremos definir como um anamorfismo de *Streams* é do tipo

$$fair_merge' :: (\text{Stream } a, \text{Stream } a) + (\text{Stream } a, \text{Stream } a) \rightarrow \text{Stream } a$$

podemos representar o anamorfismo *fair_merge'* através do seguinte diagrama:



Com a análise deste diagrama, e do comportamento da função *fair_merge* definida de forma mutuamente recursiva na formulação do problema neste enunciado

$$\begin{aligned}
& fair_merge :: (\text{Stream } a, \text{Stream } a) + (\text{Stream } a, \text{Stream } a) \rightarrow \text{Stream } a \\
& fair_merge = [h, k] \text{ where} \\
& \quad h(\text{Cons}(x, xs), y) = \text{Cons}(x, k(xs, y)) \\
& \quad k(x, \text{Cons}(y, ys)) = \text{Cons}(y, h(x, ys))
\end{aligned}$$

conseguimos definir $fair_merge'$ como um anamorfismo de *Streams*

$$fair_merge' = \llbracket geneFM \rrbracket$$

onde o gene $geneFM$ é definido como:

$$\begin{aligned} geneFM :: Stream a, Stream a &+ (Stream a, Stream a) \rightarrow (a, (Stream a, Stream a) + (Stream a, Stream a)) \\ geneFM(i_1(Cons(x, xs), y)) &= (x, i_2(xs, y)) \\ geneFM(i_2(x, Cons(y, ys))) &= (y, i_1(x, ys)) \end{aligned}$$

Para comprovar este resultado, podemos utilizar a lei dual da recursividade mútua que foi demonstrada anteriormente. Seja $fair_merge$ dada por:

$$fair_merge = [h, k]$$

E querendo demonstrar que $fair_merge = \llbracket geneFM \rrbracket$, onde $geneFM$ pode ser decomposto em:

$$geneFM = [g1, g2]$$

Tem-se:

$$\begin{aligned} [h, k] &= \llbracket geneFM \rrbracket \\ \equiv & \quad \{ \text{Lei dual da recursividade mútua} \} \\ \left\{ \begin{array}{l} \text{out} \cdot h = (id \times [h, k]) \cdot g1 \\ \text{out} \cdot k = (id \times [h, k]) \cdot g2 \end{array} \right. \\ \square & \end{aligned}$$

Para que estas equações sejam verdadeiras, é necessário encontrar $g1$ e $g2$ tais que as igualdades se verifiquem.

Vamos definir $g1$ e $g2$ como:

$$\begin{aligned} g1(Cons(x, xs), y) &= (x, i_2(xs, y)) \\ g2(x, Cons(y, ys)) &= (y, i_1(x, ys)) \end{aligned}$$

Para a primeira equação, temos:

$$\begin{aligned} \equiv & \quad \{ \text{Igualdade extensional} \} \\ & \text{out} \cdot h(Cons(x, xs), y) \\ \equiv & \quad \{ \text{Def-comp} \} \\ & \text{out}(h(Cons(x, xs), y)) \\ \equiv & \quad \{ \text{Def-h} \} \\ & \text{out}(Cons(x, k(xs, y))) \\ \equiv & \quad \{ \text{Def-out} \} \\ & (x, k(xs, y)) \\ \equiv & \quad \{ \text{Def-x, Cancelamento-+} \} \end{aligned}$$

$$\begin{aligned}
& (id \times [h, k]) (x, i_2 (xs, y)) \\
\equiv & \quad \{ \text{Def-g1} \} \\
& (id \times [h, k]) \cdot g1 (\text{Cons} (x, xs), y) \\
\square
\end{aligned}$$

O mesmo aplica-se para a segunda equação:

$$\begin{aligned}
& \equiv \quad \{ \text{Igualdade extensional} \} \\
& \text{out} \cdot k (x, \text{Cons} (y, ys)) \\
\equiv & \quad \{ \text{Def-comp} \} \\
& \text{out} (k (x, \text{Cons} (y, ys))) \\
\equiv & \quad \{ \text{Def-k} \} \\
& \text{out} (\text{Cons} (y, h (x, ys))) \\
\equiv & \quad \{ \text{Def-out} \} \\
& (y, h (x, ys)) \\
\equiv & \quad \{ \text{Def-x, Cancelamento-+} \} \\
& (id \times [h, k]) (y, i_1 (x, ys)) \\
\equiv & \quad \{ \text{Def-g2} \} \\
& (id \times [h, k]) \cdot g2 (x, \text{Cons} (y, ys)) \\
\square
\end{aligned}$$

Com isto fica demonstrado que *fair_merge* se pode definir como um anamorfismo de *Streams*, ou seja, $\text{fair_merge} = [(\text{geneFM})]$, para o gene *geneFM* definido como $[g1, g2]$.

Para testar a função *fair_merge'*, definimos a seguinte função auxiliar *takeStream* que permite extrair os primeiros n elementos de uma *Stream* e devolvê-los como uma lista.

$$\begin{aligned}
\text{takeStream} & :: \text{Int} \rightarrow \text{Stream } a \rightarrow [a] \\
\text{takeStream } 0 & _- = [] \\
\text{takeStream } n & (\text{Cons} (x, xs)) = x : \text{takeStream} (n - 1) xs
\end{aligned}$$

Definimos também as seguintes *Streams* de exemplo:

$$\begin{aligned}
s1 & :: \text{Stream Int} \\
s1 & = \text{Cons} (1, \text{Cons} (3, \text{Cons} (5, \text{Cons} (7, \text{Cons} (9, s1))))) \\
s2 & :: \text{Stream Int} \\
s2 & = \text{Cons} (2, \text{Cons} (4, \text{Cons} (6, \text{Cons} (8, \text{Cons} (10, s2)))))
\end{aligned}$$

A aplicação de *takeStream 10 (fair_merge' (i1 (s1, s2)))* resulta na lista:

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

refletindo a intercalação justa dos elementos das duas *Streams*, começando pela primeira.

A aplicação de *takeStream 10 (fair_merge' (i2 (s1, s2)))* resulta na lista:

$$[2, 1, 4, 3, 6, 5, 8, 7, 10, 9]$$

refletindo a intercalação justa dos elementos das duas *Streams*, começando pela segunda.

Problema 4

Introdução

De uma forma simples, o objetivo deste problema é desenhar uma função *transmitir* que descreve o comportamento de um aparelho de telegrafia avariado. Esse aparelho tenta transmitir uma mensagem palavra a palavra, mas pode falhar de forma aleatória: cada palavra pode perder-se durante a transmissão e, no final da mensagem, o envio do código "stop" também pode falhar.

Pretende-se que a função *transmitir* modele corretamente estas falhas, produzindo todas as mensagens possíveis, cada uma associada à respetiva probabilidade. Para tal, recorre-se a um catamorfismo probabilístico sobre listas, *pcataList*, cujo resultado é uma distribuição de probabilidades, representada pelo mónade *Dist*.

O comportamento local do aparelho é descrito por um *gene*, que define as decisões probabilísticas a tomar em cada passo da transmissão. A função *transmitir* = *pcataList gene* separa, assim, o mecanismo genérico de percorrer a lista, do comportamento específico do aparelho.

Antes de apresentar a solução completa, achamos útil analisar cada componente que será usado para que, de seguida, a definição do gene seja mais intuitiva de deduzir e possa ser aplicada para verificar a solução para o problema proposto.

Uso de *Dist*

Para este problema, é necessário modelar a transmissão de mensagens, em que cada palavra pode falhar de forma aleatória. Para isso, precisamos de uma forma de representar todas as mensagens possíveis e associar a cada uma a sua probabilidade de ocorrência.

O mónade *Dist* faz exatamente isso:

```
newtype Dist a = D { unD :: [(a, ProbRep)] }
```

- Cada *a* é um possível valor (no caso, uma mensagem ou palavra transmitida).
- Cada *ProbRep* é a probabilidade desse valor ocorrer (um número real entre 0 e 1, sendo a soma total das probabilidades 1).

Dist forma um mónade porque:

- Tem uma operação *return*: $\text{return } a = D [(a, 1)]$, que gera uma distribuição determinística com probabilidade 1.
- Tem uma composição de Kleisli (*bind* $\gg=$):

$$(f \bullet g) a = [(y, q * p) \mid (x, p) \leftarrow g a, (y, q) \leftarrow f x],$$

onde

$$g :: A \rightarrow \text{Dist } B$$

e

$$f :: B \rightarrow \text{Dist } C$$

são funções que representam computações probabilísticas.

Esta definição permite combinar automaticamente as probabilidades de múltiplas etapas de cálculo. Por exemplo, ao percorrer uma lista de palavras, o mónade *Dist* calcula todas as combinações possíveis de palavras transmitidas e perdidas, multiplicando as probabilidades associadas a cada passo, sem necessidade de ciclos explícitos ou cálculos manuais.

Desta forma, alterações no comportamento do *gene* propagam-se corretamente por toda a lista, sendo suficiente ajustar apenas as probabilidades associadas a cada decisão.

Uso de gene

O *gene* deverá descrever o comportamento local do aparelho de telegrafia. Para cada palavra, define quais são os resultados que podem ocorrer e com que probabilidades. Neste caso, deverá calcular o caso da perda de uma palavra ou o caso da perda da palavra final "stop".

Pode-se afirmar que o tipo do *gene* é:

$$gene :: () + (String, [String]) \rightarrow \text{Dist } [String]$$

Já que o uso do $\cdot + \cdot$ permite separar de forma natural os dois casos distintos que surgem ao percorrer uma lista:

- **Lista vazia:** representada por $i_1 ()$. Este caso corresponde ao fim da lista, permitindo decidir probabilisticamente se a transmissão termina corretamente ou se ocorre a falha de envio de "stop".
- **Lista não vazia:** representada por $i_2 (x, y)$, onde x é a cabeça da lista (palavra atual) e y é o resultado já processado da cauda. Este caso permite combinar a palavra atual com todos os resultados possíveis da cauda, aplicando as regras probabilísticas definidas.

Desta forma, o *gene* consegue tratar de forma modular e uniforme tanto o caso base da lista, quanto a transmissão de cada palavra individual, sem que a função de percorrer a lista (*pcataList*) precise de conhecer detalhes do comportamento probabilístico como: as probabilidades definidas ou possíveis decisões probabilísticas complexas (por exemplo, múltiplas opções para cada palavra ou eventos condicionais).

Uso de *pcataList*

O catamorfismo *pcataList* percorre a lista de palavras de forma recursiva, aplicando a função *gene* a cada passo. Esta abordagem permite separar a lógica de percorrer a lista, da lógica de transmissão probabilística, tornando o comportamento do aparelho mais fácil de modelar. Sabendo a declaração de *pcataList* (dada no enunciado):

$$pcataList :: ((() + (a, b)) \rightarrow \text{Dist } b) \rightarrow [a] \rightarrow \text{Dist } b$$

E que em listas, os catamorfismos devem ter em conta os casos de lista vazia e não vazia, é possível deduzir a definição de *pcataList* como sendo:

$$\begin{aligned} pcataList \text{ } gene \text{ } [] &= gene \text{ } (i_1 \text{ } ()) \\ pcataList \text{ } gene \text{ } (x : xs) &= \textbf{do} \\ &\quad y \leftarrow pcataList \text{ } gene \text{ } xs \\ &\quad gene \text{ } (i_2 \text{ } (x, y)) \end{aligned}$$

Nesta definição:

- No caso da lista ser vazia, *gene* recebe $i_1 ()$, permitindo decidir probabilisticamente se o processo termina.
- No caso da lista não ser vazia, a cauda da lista é processada primeiro recursivamente, produzindo y , que é então combinado com a cabeça x através de *gene* ($i_2 (x, y)$).

Para ilustrar o seu funcionamento, considere a mensagem:

`["hi","hi again","bye"]`

A aplicação de *pcataList gene* a esta lista é definida recursivamente, seguindo a estrutura do catamorfismo sobre listas. Os passos da execução, são:

- **Passo 1: lista completa ["hi","hi again","bye"]**

$$\begin{aligned} pcataList\ gene\ ["hi", "hi\ again", "bye"] &= \textbf{do} \\ y \leftarrow pcataList\ gene\ ["hi\ again", "bye"]; \\ gene\ (i_2\ ("hi", y)) \end{aligned}$$

O valor *y* representa uma distribuição de todos os resultados possíveis da cauda ["hi again","bye"].

- **Passo 2: cauda ["hi again","bye"]**

$$\begin{aligned} pcataList\ gene\ ["hi\ again", "bye"] &= \textbf{do} \\ y1 \leftarrow pcataList\ gene\ ["bye"]; \\ gene\ (i_2\ ("hi\ again", y1)) \end{aligned}$$

Cada resultado *y*₁ da chamada à cauda anterior é combinado com "hi again" através do *gene*, propagando todas as combinações possíveis com as probabilidades corretas.

- **Passo 3: cauda ["bye"]**

$$\begin{aligned} pcataList\ gene\ ["bye"] &= \textbf{do} \\ y2 \leftarrow pcataList\ gene\ [{}]; \\ gene\ (i_2\ ("bye", y2)) \end{aligned}$$

Cada resultado *y*₂ do caso base é combinado com "bye" através do *gene*, multiplicando as probabilidades de cada etapa.

- **Passo 4: caso base []**

$$pcataList\ gene\ [{}] = gene\ (i_1\ ())$$

Produz a distribuição inicial que representa as possíveis terminações da mensagem, de acordo com o comportamento probabilístico do *gene*.

A partir deste caso base, o catamorfismo combina sucessivamente cada palavra com todos os resultados possíveis da sua cauda, multiplicando as probabilidades locais. Deste modo, o valor

$$pcataList\ gene\ ["hi", "hi\ again", "bye"]$$

corresponde a uma única distribuição final que contém todas as mensagens possíveis resultantes da transmissão, já com as probabilidades corretamente combinadas.

Definição de *gene*

Depois de analisados os três componentes principais desta solução – o mónade *Dist*, o comportamento encapsulado pelo *gene* e o catamorfismo *pcataList* –, a definição do *gene* torna-se natural.

- Para o caso da lista ser vazia (*i*₁()), que corresponde ao fim da transmissão, podemos definir uma probabilidade de 90% de enviar corretamente a palavra final "stop" e de 10% de falhar.
- Para o caso da lista não ser vazia (*i*₂(*x*, *y*)), cada palavra *x* tem uma probabilidade de 95% de ser transmitida, enquanto que os restantes 5% representam a perda da palavra.

Assim, fica-se com:

$$\begin{aligned}
 gene :: () + (String, [String]) \rightarrow \text{Dist } [\text{String}] \\
 gene (i_1 ()) &= D [(["stop"], 0.9), ([], 0.1)] \\
 gene (i_2 (x, ys)) &= D [(x : ys, 0.95), (ys, 0.05)]
 \end{aligned}$$

Cálculo das soluções

Tendo $gene$ definido, basta correr no interpretador *transmitir* ["Vamos", "atacar", "hoje"] para calcular as probabilidades de todas as possíveis frases transmitidas. Fazendo isso, obtêve-se:

Mensagem transmitida	Probabilidade
["Vamos", "atacar", "hoje", "stop"]	77.2%
["Vamos", "atacar", "hoje"]	8.6%
["Vamos", "atacar", "stop"]	4.1%
["Vamos", "hoje", "stop"]	4.1%
["atacar", "hoje", "stop"]	4.1%
["Vamos", "atacar"]	0.5%
["Vamos", "hoje"]	0.5%
["atacar", "hoje"]	0.5%
["Vamos", "stop"]	0.2%
["atacar", "stop"]	0.2%
["hoje", "stop"]	0.2%
["atacar"]	0.0%
["hoje"]	0.0%
["Vamos"]	0.0%
["stop"]	0.0%
[]	0.0%

Assim, observando os resultados obtidos, é possível responder às questões propostas:

- Qual a probabilidade da palavra "atacar" da mensagem se perder? **É de 4.1%.**
- Qual a probabilidade de seguirem todas as palavras, mas faltar o "stop"? **É de 8.6%.**
- Qual a probabilidade da transmissão ser perfeita? **É de 77.2%.**

Index

\LaTeX , 5, 6
 bibtex, 6
 lhs2TeX, 5, 6
 makeindex, 6
 pdflatex, 5
 xymatrix, 7

Combinador “pointfree”

 ana, 3
 cata
 Naturais, 7
 either, 3, 4
 split, 6

Cálculo de Programas, 1, 4

 Material Pedagógico, 5
 List.hs, 4

Docker, 5

 container, 5, 6

Functor, 3, 7, 8

Função

π_1 , 7
 π_2 , 7

Haskell, 1, 5, 6

 Biblioteca

 PFP, 8

 Probability, 7, 8

 interpretador

 GHCi, 5–7

 Lazy evaluation, 3

 Literate Haskell, 5

Números naturais (\mathbb{N}), 7

Programação

 literária, 5, 6

References

- [1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [2] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 131–136. ACM, 2000.
- [3] J.N. Oliveira. Program Design by Calculation, 2024. Draft of textbook in preparation. First version: 1998. Current version: Sep. 2024. Informatics Department, University of Minho ([pdf](#)).